

# Global Value Numbering: A Precise and Efficient Algorithm

Rekha R. Pai

National Institute of Technology Calicut, Kerala, India  
rekharamapai@nitc.ac.in

**Abstract.** Global Value Numbering (GVN) is an important static analysis to detect equivalent expressions in a program. We present an iterative data-flow analysis GVN algorithm in SSA for the purpose of detecting total redundancies. The central challenge is defining a *join* operation to detect equivalences at a join point in polynomial time such that later occurrences of redundant expressions could be detected. For this purpose, we introduce the novel concept of *value  $\phi$ -function*. We claim the algorithm is precise and takes only polynomial time.

**Keywords:** Global Value Numbering, redundancy detection, value  $\phi$ -function

## 1 Introduction

Global Value Numbering is an important static analysis to detect equivalent expressions in a program. Equivalences are detected by assigning *value numbers* to expressions. Two expressions are assigned the same value number if they could be detected as equivalent. The seminal work on GVN by Kildall [1] detects all *Herbrand equivalences* [2] in non-SSA form of programs using the powerful concept of *structuring* but takes exponential time. Efforts were made to improve on efficiency in detecting equivalences. However the algorithms are either as precise as Kildall's [3] or efficient [2, 4, 5] but not both.

The strive for combining precision with efficiency has motivated our work in this area. We propose an iterative data-flow analysis GVN algorithm to detect redundancies in SSA form of programs that is precise as Kildall's and efficient (i.e. take only polynomial time). As in a data-flow analysis problem, the central challenge is to define a *join* operation to detect all equivalences at a join point in polynomial time such that any later occurrences of redundant expressions could be detected. We introduce the novel concept of *value  $\phi$ -function* for this purpose.

## 2 Terminology

*Program Representation* Input to our algorithm is the Control Flow Graph (CFG) representation of a program in SSA. The graph has empty *entry* and *exit* blocks. Other blocks contain assignment statements of the form  $x = e$ ,

where  $e$  is an expression which is either a constant, a variable, or of the form  $x \oplus y$  such that  $x$  and  $y$  are constants or variables and  $\oplus$  is a generic binary operator. An expression can also be of the form  $\phi_k(x, y)$ , called  $\phi$ -functions, where  $x$  and  $y$  are variables and  $k$  is the block in which it appears. We assume a block can have at most two predecessors and a block with exactly two predecessors is called *join* block. The input and output points of a block are called *in* and *out* points, respectively, of the block. The *in* point of a join block is called *join point*. We may omit the subscript  $k$  in  $\phi_k$  when the join block is clear from the context. In the CFGs we draw,  $\phi$ -functions appear in join blocks. But for clarity in explaining some of our concepts we assume  $\phi$ -functions are transformed to *copy* statements and appended to appropriate predecessors of the join block.

*Equivalence* Two expressions  $e_1$  and  $e_2$  are *equivalent*, denoted  $e_1 \equiv e_2$ , if they will have the same value whenever they are executed. Two expressions in a path are said to be *equivalent in the path* if they are equivalent in that path. We detect only Herbrand equivalences [2] which is equivalence among expressions with same operators and corresponding operands being equivalent.

### 3 Basic Concept

Our main goal is to detect equivalences with a view to detecting redundancies in a program in polynomial time. We introduce the concept of *value  $\phi$ -function* for the purpose which is explained in this section followed by our method to detect redundancies.

#### 3.1 Value $\phi$ -function

Consider the simple code segment in Fig.1(a). Here irrespective of the path taken  $x_1 + y_1$  is equivalent to  $a_1 + b_1$ . In terms of the variables being assigned to, we can say  $z_1$  is equivalent to same variable  $c_1$ . Now consider the code segment

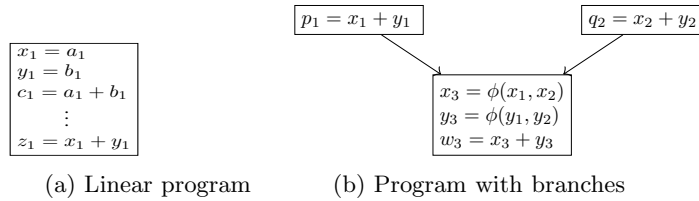


Fig. 1: Concept of value  $\phi$ -function

in Fig.1(b). Depending on the path taken expression  $x_3 + y_3$  is equivalent to either  $x_1 + y_1$  or  $x_2 + y_2$ . In terms of the variables being assigned to, we can say  $w_3$  is equivalent to merge of different variables –  $p_1$  and  $q_2$ . Inspired by the

notion of  $\phi$ -function, we can say  $w_3$  is equivalent to  $\phi(p_1, q_2)$ . This notion of  $\phi$ -function is an extended notion of  $\phi$ -function as seen in the literature. In the literature, a  $\phi$ -function has different subscripted versions of the same non-SSA variable, say  $\phi(x_1, x_2)$ . To express such equivalences, we introduce the concept of *value  $\phi$ -function* similar to the concept of *value expression* [3].

*Value  $\phi$ -function* A *value  $\phi$ -function* is an abstraction of a set of equivalent  $\phi$ -functions (including the extended notion of  $\phi$ -function). Let  $v_i, v_j$  be value numbers and  $vpf$  be a value  $\phi$ -function. Then  $\phi_k(v_i, v_j)$ ,  $\phi_k(vpf, v_j)$ ,  $\phi_k(v_i, vpf)$ , and  $\phi_k(vpf, vpf)$  are *value  $\phi$ -functions*.

*Partition* A partition at a point represents equivalences that hold in the paths to the point. An equivalence class in the partition has a value number and elements like variables, constant, and value expression. It is also annotated with a value  $\phi$ -function when necessary. The notation for a partition is similar to that in [3] except that a class can be annotated with value  $\phi$ -function.

## 4 Proposed Method

Using the concept of value  $\phi$ -function we propose an iterative data-flow analysis algorithm to compute equivalences at each point in the program. The two main tasks in this algorithm are *join operation* and *transfer function*:

### 4.1 Join operation.

A *join operation* detects equivalences that are common in all paths to a join point. The join is conceptually a class-wise intersection of input partitions. Let  $C_1$  and  $C_2$  be two classes, one from each input partition. If the classes have same value number then the resulting class  $C$  is intersection of  $C_1$  and  $C_2$ . If the classes have different value numbers, say  $v_1$  and  $v_2$  respectively, then common equivalences are found by intersection of  $C_1$  and  $C_2$ . The common equivalences obtained are actually a merge of different variables, which is indicated by the difference in value numbers and hence class  $C$  is annotated with  $\phi(v_1, v_2)$ . Now if the classes have different value expressions, say  $v_m + v_n$  and  $v_p + v_q$  respectively, the value expressions may be merged to form a resultant value expression say  $v_i + v_j$ . Value expressions  $v_m + v_n$  and  $v_p + v_q$  are merged to get  $v_i + v_j$  by recursively merging classes of  $v_m$  and  $v_p$  to get class of  $v_i$  and classes of  $v_n$  and  $v_q$  to get class of  $v_j$  [3]. But merging the value expressions can lead to exponential growth of resulting partition [5]. We do not merge different value expressions now instead merge them at a point where an expression represented by  $v_i + v_j$  actually occurs in the program. This merge is achieved simply by detecting equivalence of  $v_i + v_j$  with  $\phi(v_1, v_2)$  and is done during application of transfer function.

*Example* Let us now consolidate the concept of join using an example. Consider the case of applying join on partitions  $P_1 = \{v_1, x_1, x_3 | v_2, y_1, y_3, v_1 + 1 | v_3, z_1, z_3\}$  and  $P_2 = \{v_4, x_2, x_3 | v_5, y_2, y_3 | v_6, z_2, z_3, v_4 + 1\}$ . In the classes with value numbers  $v_1$  in  $P_1$  and  $v_4$  in  $P_2$  there is only one common variable  $x_3$  and this will appear in a class in the resulting partition  $P_3$ . Since the two classes in  $P_1$  and  $P_2$  have different value numbers  $v_1$  and  $v_4$ , respectively, the resulting class is annotated with value  $\phi$ -function  $\phi(v_1, v_4)$ . The class is assigned a new value number, say  $v_7$ . The resulting class is  $|v_7, x_3 : \phi(v_1, v_4)|$ . Now consider the classes with value numbers  $v_2$  in  $P_1$  and  $v_6$  in  $P_2$ . There are no obvious common equivalences in the classes and we don't merge the different value expressions now. Hence no new class is created. Similar strategies are adopted in detecting common equivalences in other pairs of classes one each from  $P_1$  and  $P_2$ . The resulting partition  $P_3$  is  $\{v_7, x_3 : \phi(v_1, v_4) | v_8, y_3 : \phi(v_2, v_5) | v_9, z_3 : \phi(v_3, v_6)\}$ .

```

JOIN( $P_1, P_2$ )
1   $P = \{\}$ 
2  for each pair of classes  $C_i \in P_1$  and  $C_j \in P_2$ 
3       $C_k = C_i \cap C_j$                                 // set intersection
4      if  $C_k \neq \{\}$  and  $C_k$  does not have value number
5          then  $C_k = C_k \cup \{v_k, \phi_b(v_i, v_j)\}$         //  $v_k$  is new value number
                                                    //  $v_i \in C_i, v_j \in C_j, b$  is join block
6       $P = P \cup C_k$                                      // Ignore when  $C_k$  is empty
return  $P$ 

```

Note: We define special partition  $\top$  such that  $\text{JOIN}(\top, P) = P = \text{JOIN}(P, \top)$ . We assume  $\phi$ -functions in a join block are transformed to copies and appended to appropriate predecessors of join block.

## 4.2 Transfer Function.

Given a partition  $PIN_s$ , that represents equivalences at *in* point of a statement  $s : x = e$  the transfer function computes equivalences at its *out* point, denoted  $POUT_s$ . Let  $ve$  be the value expression of  $e$  computed using  $PIN_s$ . If  $ve$  is present in a class in  $PIN_s$ , then  $x$  is just inserted into corresponding class in  $POUT_s$ . Otherwise the transfer function checks whether  $e$  could be expressed as a merge of variables represented by a value  $\phi$ -function  $vpf$  (as illustrated below). If it is present in a class in  $PIN_s$  then  $x, ve$  are inserted into corresponding class in  $POUT_s$ . Else a new class is created in  $POUT_s$  with new value number and  $x, ve, vpf$  are inserted into it.

For an example, consider processing the statement  $w_3 = x_3 + y_3$  as shown in code segment in Fig. 2. Since value expression  $v_7 + v_8$  of  $x_3 + y_3$  is not in  $PIN_3$ , the transfer function proceeds to check whether  $x_3 + y_3$  is actually a merge of variables as follows:

$$x_3 + y_3 \equiv v_7 + v_8 \equiv \phi(v_1, v_4) + \phi(v_2, v_5) \equiv \phi(v_1 + v_2, v_4 + v_5) \equiv \phi(v_3, v_6).$$

This implies  $x_3 + y_3$  is actually a merge of variables, here  $p_1$  and  $q_2$ . Since neither

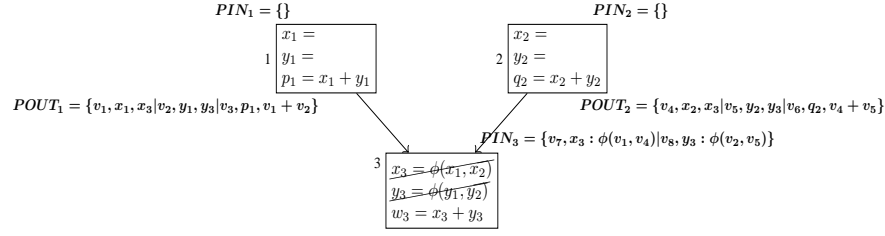


Fig. 2: Concept of Transfer Function

$v_7 + v_8$  nor  $\phi(v_3, v_6)$  are present in  $PIN_3$ , a new class is created in  $POUT_3$  with new value number say  $v_9$  and  $w_3$ ,  $v_7 + v_8$ , and  $\phi(v_3, v_6)$  are inserted into it. The classes in  $PIN_3$  are inserted as such into  $POUT_3$ . The resulting partition  $POUT_3$  is  $\{v_7, x_3 : \phi(v_1, v_4) | v_8, y_3 : \phi(v_2, v_5) | v_9, w_3, v_7 + v_8 : \phi(v_3, v_6)\}$ .

TRANSFERFUNCTION( $x = e, PIN_s$ )

```

1   $POUT_s = PIN_s$ 
2   $C_i = C_i - \{x\}$                                      //  $x \in C_i$ , a class in  $POUT_s$ 
3   $ve = \text{VALUEEXPR}(e)$ 
4   $vpf = \text{VALUEPHIFUNC}(ve, PIN_s)$                      // can be NULL
5  if  $ve$  or  $vpf$  is in a class  $C_i$  in  $POUT_s$              // ignore  $vpf$  when NULL
6      then  $C_i = C_i \cup \{x, ve\}$                          // set union
7      else  $POUT_s = POUT_s \cup \{v_n, x, ve : vpf\}$      //  $v_n$  is new value number
      return  $POUT_s$ 

```

The VALUEPHIFUNC is a recursive algorithm to compute value  $\phi$ -function corresponding to input value expression when possible else it returns NULL.

### 4.3 Detect Redundancies.

Given partition  $POUT$  at *out* of statement  $x = e$ , expression  $e$  is detected to be redundant if there exists a variable in the class of  $x$  in  $POUT$ , other than  $x$ , or the class of  $x$  in  $POUT$  is annotated with value  $\phi$ -function. In the example code in Fig. 2, consider the case of checking whether  $x_3 + y_3$  in the last statement  $w_3 = x_3 + y_3$  is redundant. In the class of  $w_3$  in  $POUT_3$  (computed in previous subsection) there are no variables other than  $w_3$ . However the class is annotated with a value  $\phi$ -function. Hence the expression  $x_3 + y_3$  is detected to be redundant.

**Theorem 1.** *Two program expressions are equivalent at a point iff the iterative data-flow analysis algorithm detects their equivalence.*

*Proof.* This can be proved by induction on the length of a path in a program.  $\square$

## 5 Complexity Analysis

Let there be  $n$  expressions in a program. The two main operations in this iterative algorithm are join and transfer function. By definitions of JOIN and TRANSFERFUNCTION a partition can have  $O(n)$  classes. If there are  $j$  join points, the total time taken by all the join operations in an iteration is  $O(n.j)$ . The transfer function involves constructing and then looking up for value expression or value  $\phi$ -function in the input partition. The transfer function of a statement takes  $O(n.j)$  time. In an iteration total time taken by transfer functions is  $O(n^2.j)$ . Thus the time taken by all the joins and transfer functions in an iteration is  $O(n^2.j)$ . In the worst case the iterative analysis takes  $n$  iterations and hence the total time taken by the analysis is  $O(n^3.j)$ .

## 6 Conclusion

We presented GVN algorithm using the novel concept of value  $\phi$ -function which made the algorithm precise and efficient.

## References

- [1] G. A. Kildall, A unified approach to global program optimization, in: Proceedings of the 1<sup>st</sup> annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages, POPL '73, ACM, New York, NY, USA, 1973, pp. 194–206.
- [2] O. Rüthing, J. Knoop, B. Steffen, Detecting equalities of variables: Combining efficiency with precision, in: A. Cortesi, G. Filé (Eds.), Static Analysis, Vol. 1694 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1999, pp. 232–247.
- [3] N. Saleena, V. Palleri, Global value numbering for redundancy detection: A simple and efficient algorithm, in: Proceedings of the 29<sup>th</sup> Annual ACM Symposium on Applied Computing, SAC '14, ACM, New York, NY, USA, 2014, pp. 1609–1611.
- [4] B. Alpern, M. N. Wegman, F. K. Zadeck, Detecting equality of variables in programs, in: Proceedings of the 15<sup>th</sup> ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '88, ACM, New York, NY, USA, 1988, pp. 1–11.
- [5] S. Gulwani, G. C. Necula, A polynomial-time algorithm for global value numbering, Science of Computer Programming 64 (1) (2007) 97–114, special issue on the 11<sup>th</sup> Static Analysis Symposium - {SAS} 2004.